Sound and Efficient Concurrency Bug Prediction

Yan Cai *
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, and University of Chinese Academy of Sciences
Beijing, China
ycai.mail@gmail.com

Hao Yun
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, and University of Chinese Academy of Sciences
Beijing, China
yunhao@ios.ac.cn

Jinqiu Wang
State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, and University of Chinese Academy of Sciences
Beijing, China
wangjq@ios.ac.cn

Lei Qiao
Beijing Institute of Control Engineering
Beijing, China
fly2mars@163.com

Jens Palsberg
University of California
Los Angeles (UCLA), USA
palsberg@ucla.edu

ABSTRACT

Concurrency bugs are extremely difficult to detect. Recently, several dynamic techniques achieve sound analysis. M2 is even complete for two threads. It is designed to decide whether two events can occur consecutively. However, real-world concurrency bugs can involve more events and threads. Some can occur when the order of two or more events can be exchanged even if they occur not consecutively. We propose a new technique SeqCheck to soundly decide whether a sequence of events can occur in a specified order. The ordered sequence represents a potential concurrency bug. And several known forms of concurrency bugs can be easily encoded into event sequences where each represents a way that the bug can occur. To achieve it, SeqCheck explicitly analyzes branch events and includes a set of efficient algorithms. We show that SeqCheck is sound; and it is also complete on traces of two threads.

We have implemented SeqCheck to detect three types of concurrency bugs and evaluated it on 51 Java benchmarks producing up to billions of events. Compared with M2 and other three recent sound race detectors, SeqCheck detected 333 races in 30 minutes; while others detected from 130 to 285 races in 6 to 12 hours. SeqCheck detected 20 deadlocks in 6 seconds. This is only one less than Dirk; but Dirk spent more than one hour. SeqCheck also detected 30 atomicity violations in 20 minutes. The evaluation shows SeqCheck can significantly outperform existing concurrency bug detectors.

CCS CONCEPTS

• **Software and its engineering** → **Multithreading**; **Scheduling**; **Software testing and debugging**.

*Corresponding author

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468549

KEYWORDS

Concurrency bugs, data races, deadlocks, atomicity violations

ACM Reference Format:
Yan Cai, Hao Yun, Jinqiu Wang, Lei Qiao, and Jens Palsberg . 2021. Sound and Efficient Concurrency Bug Prediction. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468549

1 INTRODUCTION

Concurrent programs can exhibit different thread interleavings due to non-determinism, bringing concurrency bugs [33]. They can bring harmful results or even disasters [25, 32, 44].

To detect concurrency bugs, one promising direction would be a sound predictive analysis [24, 28, 29, 43, 55]. They run a concurrent program to generate traces consisting of different types of events. They then consider the dependencies of events and model such relations either as constraints [24, 28] or direct edges in graphs [43] or comparable vector clocks [29, 31, 55]. And a feasible solution or a feasible topological order or a pair of conflicting vector clocks is taken as a proof of the existence of a real concurrency bug. Different models offer different abilities and suffer from different weaknesses. For example, constraint-solver-based ones are able to check values in memory access events, producing a larger concurrency coverage; but they rely on heavy constraint solvers to guarantee their soundness and completeness. To be efficient, these techniques usually analyze a segmentation (e.g., every 10k consecutive events) of a trace [24, 28]. Graph based ones can be complete (given a trace of two threads [43]) over full traces, but are usually inefficient. Vector clock based approaches are efficient but often incomplete.

This paper focuses on efficient, sound, and complete approaches. To the best of our knowledge, M2 [43] is the state-of-the-art one. However, M2 is limited to data race prediction only; or more precisely, it is limited to predict the kinds of concurrency bugs involving two events that should occur consecutively. Data races right fall into this category because it is defined to be two conflicting memory accesses of two threads [29, 55] that occur consecutively.

Furthermore, other types of concurrency bugs such as deadlocks [6], atomicity violations [34], and order violations [33] are not limited to two events or two threads. They do not require that all involved events occur at the same time. For example, two well-protected events can form a concurrency bug if their orders are reversed [8]; however, it fits poorly with the definition of data races. It seems highly nontrivial to extend M2 to support the detection of common concurrency bugs (see Section 2).

In this paper, we address the above challenge by presenting SeqCheck, an efficient and sound tool to analyze full traces to detect various types of concurrency bugs. SeqCheck is complete when there are two threads. The core of SeqCheck is an algorithm to decide whether a sequence of two (or more) events is feasible over a given trace. Such a sequence can vary for different types of concurrency bugs but can be easily designed; it can have events from any number of threads.

SeqCheck firstly calculates an event set that is necessary for determining the feasibility of the sequence. Secondly, SeqCheck constructs a graph to reorder the events in the set. It applies four types of orders as edges including program orders, observation orders (on pairs of write and read), lock orders, and the orders from the input sequence. Finally, it computes a closure for these orders on the graph. If no cycle is found in the process, SeqCheck soundly decides that the sequence is feasible indicating a true bug.

SeqCheck is inspired by M2 [43]; however, the essential difference between them is that M2 targets deciding whether two events can be reordered one after another; whereas SeqCheck targets deciding whether a sequence of events in a specified order is feasible, no matter whether they can occur consecutively. Note, if two events can occur consecutively, they are exchangeable; but two exchangeable events may be unable to occur consecutively. As a result, M2 produces a subset of that of SeqCheck.

To achieve the above goal, SeqCheck considers branch events. Besides, for a potential concurrency bug with more than two events that have multiple ways to occur, SeqCheck natively supports a "divide-and-conquer" manner to decide it. That is, SeqCheck can decide a potential concurrency bug to be true if it decides that any sequence (corresponding to a unique way for the bug to occur) is feasible in a trace. Instead, if any two events of a sequence can occur consecutively in different traces, M2 cannot decide that the sequence can be feasible in the same trace. We show that the algorithm SeqCheck is $O(n^2 \times log(n))$ where n is the number of events. And we present a proof to show that SeqCheck is sound and is also complete when there are two threads.

We have implemented SeqCheck for Java programs to detect general concurrency bugs. We selected two sets of previously used Java benchmarks with 31 from [3, 37, 43] and 20 from [27, 28]. They produced traces up to millions or even billions of events. On detecting data races and atomicity violations, we compared SeqCheck with (1) M2 and other three sound algorithms SHB, WCP, SyncP [37] and with AtomFuzzer [41] on the first set of benchmarks, respectively. On detecting deadlocks, we compared SeqCheck with (2) Dirk (a sound deadlock prediction tool) on the second set of benchmarks.

The experiment shows that SeqCheck significantly outperformed others on both effectiveness and efficiency. On race detection, SeqCheck detected 333 races in 30 minutes; the others detected from 130 to 285 races in at least 6 hours. The latter four reached our time limit (1 hour) on almost all large-scale benchmarks. On deadlock detection, SeqCheck detected 20 ones in 6 seconds; this number is only 1 less than that by the constraint-solver-based Dirk (that are expected to detect more than ours). However, Dirk spent >1 hour. SeqCheck detected 30 atomicity violations in the first set of benchmarks whereas AtomFuzzer detected none or crashed.

In summary, we make the following contribution:

- We propose a dynamic approach SeqCheck that models program branches and predicts the feasibility of event sequences. Thus, we turn the detection of concurrency bugs into a question of feasibility of an event sequence. And we propose how to detect three types of concurrency bugs.
- We present an analysis to show that SeqCheck is sound and is also complete when there are only two threads, and further show SeqCheck has a time complexity of $O(n^2 \times log(n))$.
- We have implemented SeqCheck and compared it with several recent sound works. An experiment confirms that SeqCheck is significantly more efficient and effective than others.

2 PRELIMINARIES AND MOTIVATIONS

2.1 Basic Definitions

This section describes a set of definitions and notations about sequentially consistency memory models [30] that are similar to definitions found in previous papers [28, 29, 43] .

Execution trace. An (execution) trace σ represents a linearization of a multithreaded program execution. It is a totally ordered list of its events, for which the order is denoted by $<_\sigma$. For a trace σ, we use $T(\sigma)$ to denote the number of threads in trace σ, and use σ_t to denote the projection of σ on thread $t \in T(\sigma)$. Each event $e \in \sigma$ has a thread ID and a event ID, which can be extracted by $tid(e)/eid(e)$. $tid(e)$ denotes the thread which e belongs. $eid(e)$ denotes the index of e in $\sigma_{tid(e)}$.

There are three categories of events (other synchronization events can be handled similarly [24, 28, 43]):

- Memory event: **write/read**, denoted by $wr(t, x)/rd(t, x, w)$, indicates a thread t writes to a (memory) location x, or read from a location x where the last write event to x is w and w can be $\varnothing$.
- Lock event: **acquire/release**, denoted by $acq(t, l)$ / $rel(t, l)$, indicates a thread t acquires or releases a lock l. Other implicit synchronizations can be treated based on this two events.
- **branch**, denoted by $br(t)$, indicates there is another path that is not followed by thread t. Note, this includes both the explicit conditional branches and the implicit branches (method calls, memory usage) in object-oriented programming languages [28].

We denote the set of event types as $\{wr, rd, acq, rel, br\}$ and use $op(e)$ to extract the type of an event e. We suppose that each thread starts and ends with a branch event $\varnothing_S$ and $\varnothing_E$, respectively.

In the rest of this paper, we may omit the thread ID of an event or the write event in a read event if there is no ambiguity in the context. We assume that lock acquire/release events are well-nested, i.e., if a thread has acquired multiple locks at a time, the corresponding lock release events must be in the nested manner.

We define a set of auxiliary functions. For a memory/lock event e, we use $loc(e)$ to get its location/lock. We denote the set of all locations of a trace σ as L_σ. For a read event e, we use $obs_\sigma(e)$ and $obs_X(e)$ to denote the involved write event in a trace σ or a set of events X. For a lock acquire/release event e, we use $match_\sigma(e)$ to denote the corresponding paired lock release/acquire event. We use $last_x^{op}(e)$ and $next_x^{op}(e)$ to denote the most recent event that operates on x before and after e in program order, respectively, where $op \in \{wr, rd, acq, rel, br\}$, where x and op can be omitted indicating any event type and any location, respectively.

We use $\mathcal{E}_\sigma$ and $\mathcal{E}_X$ to denote all events in a trace σ and all events in a set X, respectively. And we use $\mathcal{E}_X^{rd}, \mathcal{E}_X^{wr}, \mathcal{E}_X^{acq}, \mathcal{E}_X^{rel}$ to denote the set of all read events, all write events, all lock acquire events, and all lock release events in X, respectively. We define the [] operation on them as the projection on a location/lock, e.g., $\mathcal{E}_X^{rd}[x] = \{e \in \mathcal{E}_X^{rd} \mid loc(e) = x\}$.

Two events e_1 and e_2 from different threads are **conflicting**, denoted as $e_1 \propto e_2$ if: (1) they are memory events on the same location and at least one of them is a write event; or (2) they are lock events and have the same lock. We use $ConfSet(X, e)$ to find the conflicting event of e in an event set X (We only use this function when there is only one conflicting event in X).

For simplicity, we view a sequence of events $\rho = \langle e_1, e_2, \cdots e_n \rangle$ as an array (e.g., $\rho[1]$ refers to e_1). A sequence ρ' is a **read variant** of another sequence ρ, denote as $\rho \simeq \rho'$ if $|\rho| = |\rho'|$ and, for $1 \le i \le |\rho|$, we have either $\rho[i] = \rho'[i]$ or $\rho[i] = rd(t, x, w) \wedge \rho'[i] = rd(t, x, w')$. A sequence ρ is **w-r consistent** if, for any its read event $e = rd(t, x, w)$, w is identical to the most recent write event on x before e in ρ. That is, a read event always reads a value from the latest write event to the location. A prefix of a sequence $\rho = \langle e_1, e_2, \cdots e_n \rangle$ is a sequence $\rho' = \langle e_1, e_2, \cdots e_i \rangle$ where $1 \le i \le n$ or $\rho' = \emptyset$. We denote the set of all prefixes of ρ as $prefix(\rho)$. Note that a trace σ is also regard as an event sequence.

2.2 Orders

Given a trace σ, we define three basic types of orders:

- **Program order** $\prec_{PO}$. $\forall e_1, e_2 \in \mathcal{E}_\sigma : tid(e_1) = tid(e_2) \wedge e_1 \prec_\sigma e_2 \Rightarrow e_1 \prec_{PO} e_2$ (i.e., among thread local events).
- **Observation order** $\prec_{OO}$. Let $X = \mathcal{E}_\sigma, \forall e \in \mathcal{E}_X^{rd}$: $e = rd(t, x, w) \Rightarrow w \prec_{OO} e$.
- **Lock order** $\prec_{LO}$. Let $X = \mathcal{E}_\sigma, \forall e_1, e_2 \in \mathcal{E}_X^{acq}$: $e_1 \propto e_2 \Rightarrow match_\sigma(e_1) \prec_{LO} e_2 \vee match_\sigma(e_2) \prec_{LO} e_1$.

2.3 Motivations

M2 [43] is a sound predictive technique for race detection and is also complete when there are only two threads. For a pair of conflict events (e_1, e_2), M2 firstly builds a graph where vertexes are events that may affect the execution of e_1 and e_2 and edges are defined as three types of orders. M2 then applies a closure algorithm on the graph. After that, if there is no cycle formed, M2 decides that the two events can occur at the same time and they form a race. Otherwise, it makes no decision (unless the trace has two threads and, in this case, M2 decides that the two events do not form a race).

M2 decides whether two events can occur consecutively to detect races. Even if it can be extended to further check whether more events from different threads can occur consecutively, detecting common concurrency bugs requires to determine whether an order of two or more events can be reordered. This cannot be resolved by M2. Let's discuss this point.

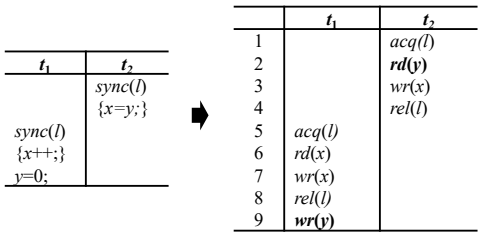

Figure 1: A trace (right) of two threads (left).

Figure 1 shows two threads and one trace where thread t_1 executes after thread t_2. Let's denote each event in the trace by their line numbers. M2's purpose is to check whether e_2 and e_9 is a race. M2 firstly computes a set of dominating orders. For e_2, the set is empty as no event before it can affect the execution of e_2. For e_9, event e_6 dominates it where e_6 reads a value on x written by e_3. Similarly, we have that e_3 is dominated by e_2. As a result, the set dominating e_9 contains e_2. This indicates that the two cannot execute consecutively. M2 decides that they do not form a race. Obviously, this conclusion is false given that the trace stems from the code on the left.

Next, suppose that M2 is extended to determine the orders, say whether the execution order from e_2 to e_9 can be reversed into that from e_9 to e_2 in an alternative execution. Obviously, this targeted order and the concluded dominating order (i.e., e_2 dominates e_9) together form a cycle. As a result, M2 decides that the target order cannot be reversed. However, it is obvious again that the target (from e_9 to e_2) order is feasible in a different trace.

The reason failing M2 on the above two examples is at its execution model. M2 follows the model [29, 36, 55] that requires: every read event in an inferred (partial) execution should read a value written by the same write event as the original trace; any other inferred (partial) execution violating it is unsound (i.e., not guaranteed to be feasible). That is, the model implicitly assumes that any read event is followed by a branch; and reading a value from a different write event may produce execution divergence.

To make M2 workable in the above example, *branches* must be explicitly considered. Actually, some constraint-based approaches already consider branches [24, 28] where they require: a read event should read the same value (that can be from different events) as that in the original execution, and any violation to it may produce an infeasible trace. This results in a huge search space for large-scale programs and constraint-solvers can be inefficient on them [24, 28]. Besides, it is difficult to consider constraints (e.g., the logic operations like "OR") for graph-based approaches like M2.

Suppose M2 is adapted to recognize branch events and check the above order reversing problem. Then, there are four branches (right before and right after each of the two events) to be analyzed. One adaption for M2 is to infer races for two events through deciding whether any two branches can be executed consecutively. This requires an analysis on the four pairs of branches. However, the two are inconsistent, i.e., whether the two branches can be executed consecutively and whether the two events can form a race.

For example, Figure 2 shows a trace δ including branch events. For the two events e_{12} and e_6, the adapted M2 can decide that two branches before them (i.e., e_{11} and $\varnothing_S$) cannot execute consecutively due to the write-read order from e_7 to e_{10}; moreover, the two events cannot form a race for the same reason. Now, suppose δ' is another trace which is the same as δ except that we swap e_6 and e_7, namely $e'_6 = wr(x)$ and $e'_7 = wr(p)$ in δ'. Now, considering the two events e_{12} and e'_7. The adapted M2 can decide that the two branches before them (i.e., e_{11} and $\varnothing_S$) cannot execute consecutively due to write-read order from e'_6 to e_{10}; but, this time, the two events indeed form a race. We can see from these two cases that there is no consistent conclusion on whether a race can be decided by deciding whether the involved branches can be executed consecutively. We can also draw the same conclusion on other cases (e.g., the branch before one event and the branch after the second event).

Another limitation is that M2 is not designed for checking multiple events from two or more threads. Say that we want to check whether the three events e_6, e_{12}, e_{18} in Figure 2 can execute in the order: e_6, e_{18}, e_{12} (this pattern can be a concurrency null pointer exception [8]) . If M2 is adapted to decide whether each two of them can form a race, it will be challenging to prove the correctness because each isolated conclusion is drawn under different conditions (e.g., write-read orders).

3 OUR APPROACH

We first present a set of definitions and then present our algorithms to check the feasibility of an event sequence. We use the trace δ and the sequence $\rho = \langle e_6, e_{18}, e_{12} \rangle$ in Figure 2 to illustrate them.

3.1 Feasible Sets

Open lock set. Given a set $X \subset \mathcal{E}_\sigma$ of a trace σ and an event $e \in \mathcal{E}_X^{acq}$, if $match_\sigma(e) \notin X$, we say e an open lock event. We use $Open(X)$ to denote the set of all open lock events in X. For example, let $X = \{e_{15-20}\}$ on trace δ; we have $Open(X) = \{e_{15}, e_{19}\}$.

Producible set (PSet). Given an event set $X \subset \mathcal{E}_\sigma$ of a trace σ and a set $Y \subseteq X$, we define a *Producible set* (or *PSet* for short) $PSet(X|Y) = P_1 \cup P_2$, where:

- $P_1 = X \setminus \mathcal{E}_Y^{rd}$, and
- $P_2 = \{e' \mid e' = rd(t, x, w') \wedge rd(t, x, w) \in \mathcal{E}_Y^{rd} \wedge w' \in \mathcal{E}_X^{wr}[x]\}$

Intuitively, $PSet(X|Y)$ have exactly the same set of events with X except that some of its read events have different but valid write events (note, in P_2, w' can be the same as that in e). In particular, if $Y = \varnothing$, we have $PSet(X|Y) = X = P_1$ and $P_2 = \varnothing$.

For the running example, let $X = \{e_{1-17}, e_{18} = rd(p, e_{12})\}$ and $Y = \{e_{18} = rd(p, e_{12})\}$ in δ; the set $X_1 = \{e_{1-17}, e'_{18} = rd(p, e_6)\}$ is a PSet of X but the set $X_2 = \{e_{1-17}, e'_{18} = rd(p, e_4)\}$ is not a PSet of X as the event e_6 is in $\mathcal{E}_X^{wr}[p] = \{e_6, e_{12}\}$ but the event e_4 is not.

The reason for introducing the set $Y \subseteq X$ is that we consider branch events explicitly. Hence, there should be a cut over a set of events such that (1) all read events in one part should read the same values as that in the original trace but (2) some read events in another part can read different values as long as they do not produce execution divergence. We will define such a Y later.

To correlate events in the two sets X and $PSet(X|Y)$, we use $s(e')$, for any event $e' \in PSet(X|Y)$, to denote its original event e in X. Notice that, for an event $e \in PSet(X|Y)$, if $e \in P_1$, then $s(e) = e$.

	t_1	t_2	t_3
1		$acq(l_2)$	
2		$wr(x)$	
3		$rel(l_2)$	
4			$wr(y)$
5			$acq(l_2)$
6			**$wr(p)$**
7			$wr(x)$
8			$rel(l_2)$
9	$acq(l_1)$		
10	$rd(x)$		
11	br		
12	**$wr(p)$**		
13	$wr(y)$		
14	$rel(l_1)$		
15		$acq(l_1)$	
16		$rd(x)$	
17		br	
18		**$rd(p)$**	
19		$acq(l_2)$	
20		$rd(y)$	
21		$rel(l_2)$	
22		$rel(l_1)$	

Figure 2: A trace δ and a sequence of events $\rho = \langle e_6, e_{18}, e_{12} \rangle$.

Given $X \subset \mathcal{E}_\sigma$ in a trace σ, let $Y \subseteq X$ and $X' = PSet(X|Y)$, we say that X' is a **Feasible Set** (or **FSet** for short) if it satisfies:

- **Program order closed** (or **prefix closed**): $\forall e'_1 \in X', \forall e_2 \in X$, if $e_2 \prec_{PO} s(e'_1)$, then $\exists e'_2 \in X' \wedge s(e'_2) = e_2$.
- **Observation feasible**: $\forall e' = rd(t, x, w') \in \mathcal{E}_{X'}^{rd}$, we have $w' \in X'$.
- **Lock feasible**: (1) $\forall e \in \mathcal{E}_{X'}^{rel}$, we have $match_\sigma(e) \in X'$ and (2) $\forall e_{acq_1}, e_{acq_2} \in \mathcal{E}_{X'}^{acq} \wedge e_{acq_1} \neq e_{acq_2}$, if $match_\sigma(e_{acq_1}) \notin X' \wedge match_\sigma(e_{acq_2}) \notin X'$, then $loc(e_{acq_1}) \neq loc(e_{acq_2})$.

The definition of FSet restricts a set to be feasible by considering program orders, observation orders, and lock orders. For the running example, let $X = \mathcal{E}_\delta$ and $Y = \{e_{4-8}, e_{12-14}, e_{18-22}\}$, then we have $X' = \{e_{1-17}, rd(p, e_6), e_{19}, rd(y, e_4), e_{21-22}\}$ is a FSet of X.

3.2 Feasible Traces

Given an execution trace σ, we say that an event sequence over $\mathcal{E}_\sigma$, denoted as σ', is a **feasible trace** if :

(1) $\sigma' \in prefix(\sigma)$, or,
(2) $\sigma' = \sigma'' \cdot e$ where σ'' is feasible and σ' is w-r consistent, and the following three conditions are satisfied:
   (a) let $t = tid(e)$, $br = last^{br}(e) \in \mathcal{E}_{\sigma'_t}$, and $\sigma'' = \sigma''' \cdot br \cdot \theta'''$, then $\exists \theta$ such that $\sigma'''_t \cdot br \cdot \theta \in prefix(\sigma_t) \wedge \theta'''_t \cdot e \simeq \theta$.
   (b) $op(e) = acq$, then $\nexists e' \in Open(\mathcal{E}_{\sigma''}) \wedge loc(e') = loc(e)$.
   (c) $op(e) = rel$, then $\exists e' \in Open(\mathcal{E}_{\sigma''_t}) \wedge loc(e') = loc(e)$.

The condition 2a requires that the appended event e must be exactly the next event of t except, if it is a read event and there is no branch event after it, it can read different but valid values.

3.3 Feasible Partial Orders

A FSet can be linearized into an event sequence. However, such a sequence is not guaranteed to be a feasible trace defined in the last subsection. This section defines a set of necessary partial orders such that, if a sequence is linearized from a FSet by reserving all partial orders over the set, then it is a feasible trace.

Given a trace σ and a partial order P over a FSet $X' = PSet(X|Y)$ where $X \subset \mathcal{E}_\sigma$ and $Y \subseteq X$, we say P is a **trace-respecting partial order** over X' if: (1) P refines the program order in σ when restricted

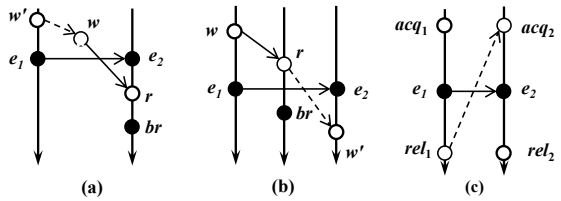

Figure 3: An illustration on the observation closure.

to events in X', (2) for every read event $r \in X'$, if $r \in X \setminus Y$ we have $obs_{X'}(r) \prec_P r$, and (3) for every lock acquire event $e_{acq} \in \mathcal{E}_{X'}$ such that $match_\sigma(e_{acq}) \notin X'$ and for every lock release event $e_{rel} \in X'$, if $e_{rel} \propto e_{acq}$, then $e_{rel} \prec_P e_{acq}$. We write that P respects X'.

Trace-closed Partial Orders. Given a trace σ and a set of events $X \subset \mathcal{E}_\sigma$, let $Y = \{e \in X | next^{br}(e) \notin X\}$ and P be a trace-respecting partial order over a feasible set of events $X' = PSet(X|Y)$. We say that P is *trace-closed* if it satisfies the following:

- **Observation-closed**. (1) For every read event $r = rd(t, x, w') \in \mathcal{E}^{rd}_{X'}$ such that $s(r) \in Y$, $\exists w' \in \mathcal{E}^{wr}_{X'}[loc(r)] \wedge w' \prec_P r$. (2) For every read event $r \in \mathcal{E}^{rd}_{X'}$ such that $s(r) \notin Y$, let $w = obs_{X'}(r)$ being conflicting with r. For every write event $w' \in \mathcal{E}^{wr}_{X'}[loc(r)] \setminus \{w\}$, (a) if $w' \prec_P r$ then $w' \prec_P w$; (b) if $w \prec_P w'$ then $r \prec_P w'$.
- **Lock-closed**. For events $\forall e_{rel_1}, e_{rel_2} \in \mathcal{E}^{rel}_{X'}$ and their matched acquire events $e_{acq_1} = match_\sigma(e_{rel_1})$ and $e_{acq_2} = match_\sigma(e_{rel_2})$, if $e_{rel_1} \propto e_{acq_2}$, then $e_{rel_1} \prec_P e_{acq_2} \vee e_{rel_2} \prec_P e_{acq_1}$.

In the definition, if there is no branch event after the read event r, the observation-closed property requires that r can happen after any write event. In other case, it requires that any write should occur either before or after both events from a pair in observation order $\langle obs_{X'}(r), r\rangle$ on the same location, as shown in Figure 3(a) and (b). This is different from that of M2 that applies same rule to all read events. Figure 3(a) illustrates that any write w' should occur before $w = obs_{X'}(r)$ as it already occurs before r (due to the order $\langle e_1, e_2\rangle$). Figure 3(b) illustrates the second case. The lock-closed property requires two conflicting critical sections not overlapping. The order of the two sections can be obtained when the order of two events in them ($\langle e_1, e_2\rangle$) is known, as illustrated in Figure 3(c). Then, we can infer that e_{rel_1} occurs before e_{acq_2}. Other cases (e_1 occurs after e_{rel_1}, e_2 occurs before e_{acq_2}) have similar results.

Given a feasible set of events $X \subset \mathcal{E}_\sigma$ from a trace σ, its FSet X' may have zero to multiple trace-closed partial orders. If it has one, we call the smallest trace-closed partial order P as the **closure** of X' and also say that P is **a feasible partial order** over X'. There exists at most one smallest trace-closed partial order, which can be proved by contradiction [43].

3.4 The SeqCheck Algorithm

This section presents the detailed algorithm of SeqCheck and also compare it with M2. SeqCheck decides the feasibility of an event sequence that encodes a potential concurrency bug. We allow two kinds of ordering to be specified by an input: (1) the orders between two events in a given sequence and (2) whether an event not from the sequence can occur in between any two events from the sequence. The second is referred to as *Adjacency* set.

Formally, given a trace σ, an event sequence $\rho = \langle e_1, e_2, ...e_n\rangle$ over a PSet, and a set of pairs $\mathcal{A} = \{(e_i, e_j) \mid e_i, e_j \in \mathcal{E}_\rho\}$, SeqCheck

Algorithm 1: ComputePotentialFSet(σ, ρ)

```
1  F ← ∅                                   /* To hold a FSet candidate. */
2  M ← ∅                                   /* To map branch events to write events. */
3  L ← ∅                                   /* To hold a set of open locks. */
4  Q ← ∅                                   /* A temp queue to keep intermediate events. */
5  foreach e ∈ E_ρ do Q.push(e)
6  for i ← |E_ρ| to 1 do
7      L' ← Open({e ∈ E_σ | e ≤_PO ρ[i]})
8      foreach e_j ∈ L' | ConfSet(L, e_j) ∉ {∅, e_j} do
9          Q.push(match_σ(e_j))
10         L' ← L'\{e_j}
11     L ← L ∪ L'
12 while ¬Q.empty() do
13     B ← ∅
14     e_cur ← Q.pop()
15     for e ∈ E_σ \ F | e ≤_PO e_cur do
16         if op(e) = br then B ← B ∪ {e}
17         else if op(e) = rd ∧ next^br(e) ≠ ∅ then
18             e_b ← next^br(e)
19             M[e_b].insert(obs_σ(e))
20         else if op(e) = acq then
21             Let e' ∈ E_ρ | ∀e'' ∈ E_ρ ∧ e'' ≤_PO e' ∧ tid(e) =
                 tid(e') ∧ tid(e) = tid(e'')
22             if e' = ∅ then Q.push(match_σ(e))
23             else if e' ≺_PO e then
24                 x ← the index of e' in ρ
25                 foreach i ← x − 1 to 1 do
26                     L' ← Open({e_j ∈ E_σ | e_j ≤_PO ρ[i]})
27                     e_c ← ConfSet(L', e)
28                     Q.push(match_σ(e_c))
29                 Q.push(match_σ(e))
30         F ← F ∪ {e}
31     foreach e_i ∈ B do
32         Q.push(M[e_i])        /* Push all events mapped from e_i. */
33         Remove key e_i from M
34     foreach e_r ∈ E^rd_F | next^br(e_r) ∉ F do
35         if no write event can be observed by e_r then
36             e_w ← the first conflicting event of e_r in thread
                 tid(obs_σ(e_r))
37             Q.push(e_w)
38 return F
```

answers whether there is a feasible trace σ' that **satisfies** ρ and $\mathcal{A}$, that is: (1) for $1 \le i < j \le n$, we have $e_i \prec_{\sigma'} e_j$ and (2) for a pair $(e_1, e_2) \in \mathcal{A}$, we have $\nexists e \in \mathcal{E}_{\sigma'}$ such that $e_1 \prec_{\sigma'} e \wedge e \prec_{\sigma'} e_2$.

Overall, SeqCheck computes a set of events as a candidate of FSet (Algorithm 1) and then checks whether there is a set of feasible partial orders over the candidate (Algorithms 2, 3, and 4); if so, the events can be reordered to obey the given sequence of events.

(1) Generate a Candidate FSet. Algorithm 1 computes a set of events F as a candidate FSet. It starts from an initial set of all events in ρ and iteratively includes additional events according to the three requirements in the definition of FSet. In the iteration, M maps a branch event to a set of write events. That is, the key of the map is a branch event and the value is a set of write events. It indicates that any read event before a branch event (in program order) read a value from one of the mapped write events. Hence, if a branch is included, all mapped write events will be included.

Additionally, for any open lock event e not from threads in ρ, the algorithm includes the release event $match_\sigma(e)$ and all other events before it by program order.

Algorithm 2: ComputeLockOrders(σ, ρ, F)

```
1  C_LO ← ∅                    /* To hold a initial set of lock semantic orders. */
2  L ← ∅   /* To keep a set of intermediate open lock acquisitions. */
3  foreach i ← |E_ρ| to 1 do
4      L' ← Open({e_j ∈ E_σ | e_j ≤_PO ρ[i]})
5      e_cur ← next(ρ[i])
6      while e_cur ∈ F ∧ L' ≠ ∅ do
7          if op(e_cur) = rel ∧ match_σ(e_cur) ≤_PO ρ[i] then
8              e_a ← match_σ(e_cur)
9              e_c ← ConfSet(L, e_a)
10             if e_c ≠ ∅ then
11                 C_LO ← C_LO ∪ {⟨e_cur, e_c⟩}
12                 L' ← L' \ {e_a}
13         else if op(e_cur) = acq ∧ ConfSet(L, e_cur) = ∅ then
14             L' ← L' ∪ {e_cur}
15         e_cur ← next(e_cur)
16     L ← L ∪ L'
17 L ← Open(F)
18 foreach e_c ∈ E_F^rel do
19     e_a = ConfSet(L, e_c)
20     C_LO ← C_LO ∪ {⟨e_c, e_a⟩}
21 return C_LO
```

Table 1: State changes of Algorithm 1 on trace δ.

Line 5	$Q = \{e_6, e_{18}, e_{12}\}, L = \{\}, F = \{\}, M = \{\}$
Line 6 – 11	$Q = \{e_6, e_{18}, e_{12}, e_{22}\}, L = \{e_9, e_5\}, F = \{\}, M = \{\}$
Line 12 – 37	$Q = \{\}, L = \{\}, F = \{e_{1-12}, e_{15-22}\}, M = \{(\varnothing_E, \{e_{12-13}\})\}$

Note, in our definition of FSet, there are two parts: X and Y. The result F is actually the X' in definition; and for any remaining key e_b (that is not removed in line 33), all read events in between e_b and its previous branch of the same threads belong to Y.

Compared to M2 that has a *RCone* algorithm to find a set of potential events starting from two given events, our algorithm focuses on a sequence of events. This brings the following major differences: (1) we include all events in ρ (line 5) to be part of F whereas M2 excludes the two events of a potential race; (2) we process all events in ρ in the reverse order as that are expected to occur (line 6) whereas M2 can start from any of two events; (3) we consider read events in two categories. For read which is followed by branch event in F, we include the corresponding write event. For the other case, we heuristically include some write, so that the read event have at least one write to observe. However, M2 treats read events the same and includes all write events if the read events are included. The three points distinguish our algorithm from that of M2. They allow us to model how branch events can affect trace feasibility and how an event sequence can be considered.

On the running example, Algorithm 1 runs as follows and we show how Q, L, F, and M changes in Table 1. Recall that each thread in δ starts and ends with a branch. The $\varnothing_E$ in Table 1 is the end branch in thread t_2. Line 5 initializes Q to include all events in ρ; next, the loop at line 6 appends e_9 into L in first iteration. When processing e_{18} in the second iteration, $e_9 \in L$ is a conflicting event of e_{15} and $match_\delta(e_{15}) = e_{22}$ is appended into Q; in the third iteration, e_5 is appended into $L = \{e_9, e_5\}$. The iteration at line 12 pops all events in Q as well as other events that occurred before them in program order; they are included into F. Finally, we have $F = \{e_{1-12}, e_{15-22}\}$.

Table 2: State changes of Algorithm 2 on trace δ.

$C_{PO} = \{\cdots\}, C_{OO} = \{\langle e_7, e_{10}\rangle, \langle e_7, e_{16}\rangle\}$, $C_{SO} = \{\langle e_6, e_{18}\rangle, \langle e_{18}, e_{12}\rangle\}$	
Line 3 – 16	$C_{LO} = \{\langle e_{22}, e_9\rangle, \langle e_8, e_{19}\rangle\}, L = \{e_9, e_{19}\}$
Line 17 – 20	$C_{LO} = \{\langle e_{22}, e_9\rangle, \langle e_8, e_{19}\rangle\}, L = \{e_9\}$

(2) Initialize a Set of Partial Orders. SeqCheck next checks whether the set F can be a FSet. It first constructs three sets of partial orders on F according to the definition of the trace-respecting partial orders: (1) program order C_{PO}, (2) observation order C_{OO}, (3) lock semantic order C_{LO}. The program order C_{PO} can be easily constructed according to the occurrence order of events from the same threads in F. The observation order C_{OO} is defined to be $\{\langle obs_F(e), e\rangle | \forall e \in \mathcal{E}_F^{rd}, next^{br}(e) \in F\}$.

The lock order C_{LO} is constructed by Algorithm 2. Unlike M2, the lock orders consist of (1) the intra-thread lock orders among (open) lock events for events in ρ (lines of the first for-loop at line 3) and (2) the lock orders between the (open) lock event for the events in ρ and for all others in F (the for-loop at line 18). The intra lock orders must be constructed by considering the sequence order.

Finally, SeqCheck includes the set of input orders C_{SO} in ρ, where $C_{SO} = \{\langle e_i, e_{i+1}\rangle | \forall i \in \{1, \cdots, |\mathcal{E}_\rho| - 1\}\}$.

Table 2 shows the four sets of orders (in its first row) given the set F. Note, although F includes the event $e_{20} = rd(y)$, there is no observation order from $obs_\delta(e_{20})$ (i.e., e_{13}) to it; this is because, by our definition, we have $next^{br}(e_{20}) \notin F$. The second row in Table 2 shows the state changes of Algorithm 2. For the example. By the algorithm, during its second iteration (the for loop at line 3), as the event e_{15} and its matching event e_{22} are both included in F, the algorithm inserts a intra-thread lock order $\langle e_{22}, e_9\rangle$. In the third iteration, the intra-thread lock order $\langle e_8, e_{19}\rangle$ is included due to that e_{18} is happened after e_6 in ρ and a conflict pair on lock l_2 exists.

(3) Compute a Closure. Given the four sets of partial orders, SeqCheck computes a closure according to the definition of the trace-closed partial order as shown in Algorithm 3. Before introducing the algorithm, we first introduce a graph data structure [43]. All partial orders will be represented as edges on such a graph.

Let G be a directed acyclic graph and X be a set of events. The vertexes of G consist of all events $\mathcal{E}_X$ and the edges are defined to be a subset of $\mathcal{E}_X \times \mathcal{E}_X$. We define a set of operations over G:

- $G.insert(e_1, e_2)$ inserts an edge $\langle e_1, e_2\rangle$ into G.
- $G.reach(e_1, e_2)$ returns True if there is a path from e_1 to e_2.
- $G.succ(e, t)$ returns the earliest successor e' of e in thread t where $G.reach(e, e')$ returns True.
- $G.pred(e, t)$ returns the latest predecessor e' of e in thread t where $G.reach(e', e)$ returns True.

These four operations over G can be done with an $O(n \times log(n))$ algorithm through Fenwick Tree [18], where $n = |\mathcal{E}_X|$.

Given a trace σ, a graph G, an initial set of orders C, and a adjacency set $\mathcal{A}$, Algorithm 3 iteratively examines each partial order in C, inserts it into G, and closes it according the definition of trace-closed partial orders. These are shown as functions *InsertAndClose*, *ObsClosure*, and *LockClosure*. For observation-closed rule, Algorithm 3 only close the rule (2). Rule (1) will be consider in Algorithm 4. Besides, it further closes any adjacency orders (lines 22–26). That is, for any pair of events $(e_1, e_2) \in \mathcal{A}$, if an event e

Algorithm 3: CloseOrders($\sigma, G, C, \mathcal{A}$)

```
1  Let F ← G.F
2  foreach x = ⟨e1, e2⟩ ∈ C do Qc.push(x)
3  while ¬Qc.empty() do
4      ⟨ex, ey⟩ ← Qc.pop()
5      if G.reach(ey, ex) then return False
6      else if ¬G.reach(ex, ey) then InsertAndClose(ex, ey)
7  return True
8  Function ObsClosure(e1, e2):
9      foreach x ∈ {loc(e) | e ∈ E_F^rd ∪ E_F^wr} do
10         e_w ← last_x^wr(e1)
11         e_r ← next_x^rd(e2)
12         if e_w ≠ obs_σ(e_r) ∧ next^br(e_r) ∈ F then
13             Qc.push(⟨e_w, obs_σ(e_r)⟩)
14         e'_w ← next_x^wr(e2)
15         foreach e_i ∈ E_F^rd | obs_σ(e_i) = e_w ∧ next^br(e_i) ∈ F do
16             Qc.push(⟨e_i, e'_w⟩)
17 Function LockClosure(e1, e2):
18     foreach l ∈ {loc(e) | e ∈ E_F^acq ∪ E_F^rel} do
19         e_acq ← last_l^acq(e1)
20         e_rel ← next_l^rel(e2)
21         Qc.push(⟨match_σ(e_acq), match_σ(e_rel)⟩)
22 Function AdjacencyClosure(A):
23     foreach (e1, e2) ∈ A do
24         foreach t ← 1 to T(σ) do
25             Qc.push(⟨G.pred(e1, t), e2⟩), Qc.push(⟨e1, G.succ(e2, t)⟩)
26             Qc.push(⟨G.pred(e2, t), e1⟩), Qc.push(⟨e2, G.succ(e1, t)⟩)
27 Function InsertAndClose(e1, e2):
28     G.insert(e1, e2)
29     ObsClosure(e1, e2)
30     LockClosure(e1, e2)
31     for i, j ← 1 to T(σ) | i ≠ tid(e1) ∨ j ≠ tid(e2) do
32         (e_pred, e_succ) ← (G.pred(e1, i), G.succ(e2, j))
33         G.insert(e_pred, e_succ)
34         ObsClosure(e_pred, e_succ)
35         LockClosure(e_pred, e_succ)
36     AdjacencyClosure(A)
```

occurs before or after one of the two event, it also occurs before or after another one. Algorithm 3 fails whenever it finds a cycle.

For the running example, the given initial set of orders are shown in Figure 4(a) except the two from events of t_2 (where we do not explicit show the program orders). The two orders are produced by Algorithm 3. Considering that $obs_\delta(e_{16})$ is e_7 and the event e_2 occurred before e_{16} by program order, Algorithm 3 reorders e_2 before $obs_\delta(e_{16})$, resulting an order $\langle e_2, e_7\rangle$. When computing the lock closure of this order, an order $\langle e_3, e_5\rangle$ is inserted.

(4) The Complete SeqCheck Algorithm. Algorithm 4 describes the complete SeqCheck. Given a trace σ, an event sequence ρ, and an adjacency set $\mathcal{A}$, it drives Algorithms 1, 2, and 3 to find a closure. The order specified by the input sequence of events ρ is denoted as C_{SO}. Then, Algorithm 4 check if the read event (which are not followed by branch event in F) has a write event to observe. If not, it heuristically let e_r observe the first conflicting write in thread $tid(obs_\sigma(e_r))$ and calculates the closure. These operations are according to the observation-closed rule (2). If it succeeds, it additionally considers all other conflicting but unordered pairs of events (line 9). Such pairs are inserted into G according to their occurrence orders in σ. After SeqCheck finishes, if there is a cycle, it returns $\emptyset$; otherwise, it returns a linearization of the graph G.

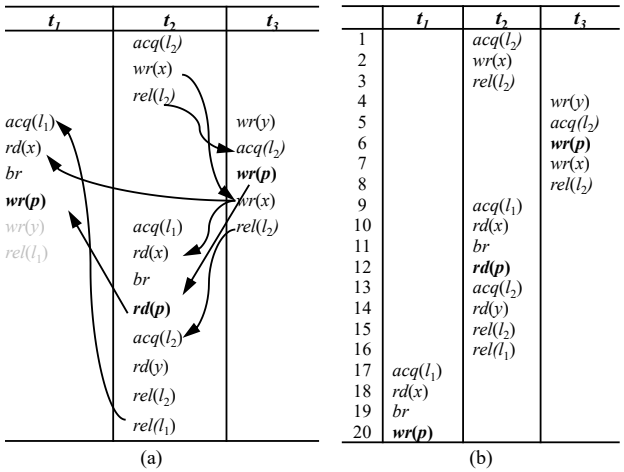

Figure 4: The closure for trace δ in (a) and one of its corresponding execution in (b).

For the running example, SeqCheck produces a set of orders as shown in Figure 4(a). As there is no cycle found and no additional event to be ordered, these orders indicate that the set F produced by Algorithm 2 is a FSet. And it decides that the input sequence ρ is feasible and Figure 4(b) shows a trace that satisfies ρ.

In summary, as reflected in Algorithms 1, 2, and 3, M2 can only handle adjacency relations, while SeqCheck can handle both adjacency relations and order relations. That is, M2 can only detect that events happen consecutively, while SeqCheck can detect that events happen in a particular order, as well as consecutively. And SeqCheck has a novel definition of feasible sets.

3.5 Algorithm Analyses

Algorithm Time Complexity. SeqCheck consists of four algorithms. Let n be the size of trace σ, the total number of events in σ. Algorithms 1 has two major loops. In the first major loop, the $Open()$ map can be initialized in $O(n)$ by scanning all events once. The function $ConfSet(L, e)$ can be implemented in $O(log(n))$. The second major loop (the *while* part) pushes each event at most once into Q, and all the operations in loop can be implemented in $O(log(n))$. So, both parts have an $O(n \times log(n))$ time complexity.

In Algorithm 2, each for/while-loop processes at most n events. For each event, there is at most a call to $ConfSet(L, e)$. That is $O(n \times log(n))$ in time complexity. Algorithm 3 computes a closure for the edges. There are at most n^2 edges and each is inserted into G once. That is $(n^2 \times log(n))$ in time complexity. Algorithm 4 has two loop, processing at most n^2 event pairs. For each pair, it inserts at most one edge. Hence, SeqCheck has $(n^2 \times log(n))$ time complexity.

Next, we give an analysis on the soundness and the completeness of SeqCheck.

Theorem 1. ***Soundness.*** *Given a trace σ, an event sequence ρ, and an adjacency set $\mathcal{A}$ over $\mathcal{E}_\rho$, if Algorithm 4 returns a linearization σ', then σ' is a feasible trace that satisfies ρ and $\mathcal{A}$.*

Proof Sketch. We show that σ' is a feasible trace by induction. Let $\sigma'' = \sigma''' \cdot e \in prefix(\sigma')$, such that σ''' is feasible (i.e., $\emptyset$ in the base step). When appending e to σ''', we show below that no condition of the definition of feasible traces is violated.

Algorithm 4: $SeqCheck(\sigma, \rho, \mathcal{A})$

```
1  F ← ComputePotentialFSet(σ, ρ)
2  C ← C_PO ∪ C_OO ∪ C_LO ∪ C_SO /* C_SO is a set of orders over ρ. */
3  G ← (F, ∅)
4  if ¬CloseOrders(σ, G, C, A) then return ∅
5  foreach e_r ∈ E_F^rd | next^br(e_r) ∉ F do
6      if ∄e_w ∈ E_F^wr [loc(e_r)] ∧ G.reach(e_w, e_r) then
7          e'_w ← the first conflicting event of e_r in thread
               tid(obs_σ(e_r))
8          if ¬InsertAndClose(e'_w, e_r) then return ∅
9  foreach unordered pair ⟨e_1, e_2⟩ ∈ G | e_1 ∝ e_2 ∧ e_1 ≺_σ e_2 do
10     if ¬InsertAndClose(e_1, e_2) then return ∅
11 σ' ← a topological order of G          /* A linearization of G */
12 return σ'
```

First, if e is a read event, Algorithm 4 initially sets an order $\langle obs_{\sigma'}(e), e\rangle$, resulting in an edge from $obs_{\sigma'}(e)$ to e in G; Algorithm 3 ensures that any other conflicting write events are ordered either before $obs_{\sigma'}(e)$ or after e. Therefore, no other conflicting write event appears in between $obs_{\sigma'}(e)$ and e in σ'. Hence, the sequence $\sigma''' \cdot e$ is w-r consistent.

Second, if e is a lock acquire event, Algorithm 1 ensures that there are no two open lock events on $loc(e)$; Algorithm 2, 3 ensures that all other conflicting lock release or acquire events are before e or after $match_\sigma(e)$. If e is not closed, they before e. The similar analysis applies when e is a lock release event. Hence, the conditions 2b and 2c in definition of feasible trace are not violated.

Third, Algorithm 4 keeps program orders; according to Algorithm 1, when an event e' is included, all other events occur before e' by program order are included. Hence, we have $\mathcal{E}_{\sigma_t''} \subset \mathcal{E}_{\sigma_t}$ and, the sequence $\sigma_t'' = \sigma_t''' \cdot e \in prefix(\sigma_t)$. And we can rewrite σ_t'' to be $\sigma_t''' \cdot br \cdot \theta$ (form in definition) that is also in $prefix(\sigma_t)$. Thus, the condition 2a is not violated.

Inductively, we show that σ' is a feasible trace when we have $|\sigma''| = |\sigma'|$.

Finally, as Algorithm 4 includes the order by ρ and Algorithm 3 closes adjacency orders according to $\mathcal{A}$, σ' satisfies ρ and $\mathcal{A}$. □

Theorem 2. ***Completeness.*** *Given a trace σ of two threads, an event sequence ρ, and an adjacency set $\mathcal{A}$ over $\mathcal{E}_\rho$, if there is a feasible trace σ' that satisfies ρ and $\mathcal{A}$, Algorithm 4 returns an event sequence.*

Proof Sketch. When there are two threads, the initial sets $C_{PO} \cup C_{OO} \cup C_{LO} \cup C_{SO}$, their closure by Algorithm 3, the closure on $\mathcal{A}$ are necessary orders to witness ρ.

If no cycle forms after Algorithm 4 in line 4, SeqCheck will check the read events which are not followed by branch event one by one. Because there are only two threads in σ, a read event can only observe the write event (1) before it in program order, or (2) in the other threads. When the line 8 need to be executed, it must meet the second case. In other words, there must be no relative write event before it in the same thread. So, it's a definite case to add the order, from the first conflicting write event of the other thread to the read event. It's a necessary order.

Then, SeqCheck checks others unordered pairs. In this procedure, only some critical sections are unordered. This procedure will not form cycle, as any unordered critical section indicates that there are no conflicting event pair to dominate two section. (Actually, ordering them is useful for linearization.)

Finally, Algorithm 4 returns a linearization of G, i.e., an event sequence. □

Algorithm 5: DetectConcurrencyBugs(σ)

```
   /* Detect data races                                              */
1  foreach pair of conflicting memory events (e_1, e_2) do
2      ρ ← ⟨last(e_1), e_2, e_1, next(e_2))⟩
3      A ← {(e_1, e_2)}
4      if SeqCheck(σ, ρ, A) ≠ ∅ then
5          print "A data race detected"
   /* Detect deadlocks of two threads                                */
6  foreach potential deadlock (⟨acq_l^1, acq_m^1⟩, ⟨acq_m^2, acq_l^2⟩) do
7      ρ_1 ← ⟨acq_m^2, acq_m^1, acq_l^2⟩
8      ρ_2 ← ⟨acq_l^1, acq_l^2, acq_m^1⟩
9      if SeqCheck(σ, ρ_1, ∅) ≠ ∅ ∨ SeqCheck(σ, ρ_2, ∅) ≠ ∅ then
10         print "A deadlock detected"
   /* Detect atomicity violations of the pattern "w – w – r"         */
11 foreach potential atomicity violation: (w_1, w_2, r) do
12     ρ ← ⟨w_2, w_1, r⟩
13     if SeqCheck(σ, ρ, ∅) ≠ ∅ then
14         print "An atomicity violation detected"
```

Note, if there are additional threads, there may have conflicting critical sections that will be included by Algorithm 1. Orders among these events may not be necessary to witness a sequence and a cycle may be introduced.

4 IMPROVE PERFORMANCE

SeqCheck can suffer from an overhead that stems from handling a large number of orders and event pairs in searching for any unordered events (that are part of potential bugs). M2 adopts an optimization to only consider pairs of conflicting events that are neither protected by common locks nor ordered by trace-respecting partial orders. The optimization is in a pre-process phase under an $O(n^2 \times log(n))$ (where n is the number of events) algorithm.

We also include a pre-process phase. However, we construct a graph G' to have an initial set of program orders and observation orders and then to compute a transitive closure (with considering branches) on G'. Then from G', we can easily identify pairs of already ordered events; and the remaining pairs are undecided. The algorithm to construct the graph is $O(k^2 \times n \times log(n))$ in time complexity, where k and n are the number of threads and the number of events, respectively. Moreover, pre-storing the observation orders and performing an de-duplication are also an acceleration.

5 DETECT GENERAL CONCURRENCY BUGS

This section presents Algorithm 5 that drives Algorithm 4 to detect three types of concurrency bugs by encoding them into sequences of events. Other types can be implemented similarly.

Detect Races. A race occurs when two conflicting events occur consecutively. Our work decides sequence of events and cannot be directly used to detect races. Algorithm 5 introduces two more events $last_\sigma(e_1)$ and $next_\sigma(e_2)$. (Note, these two events can be dummy ones as long as they are right before and right after $e_{1/2}$ by program order.) Besides, there is an adjacency ordering

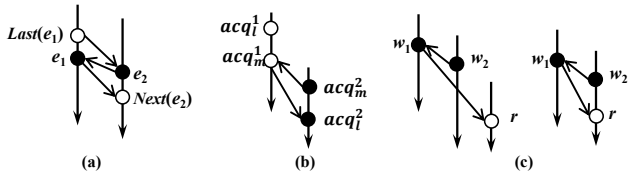

Figure 5: An illustration of part of sequence for three concurrency bugs.

requirement: no other event should be between e_1 and e_2. It then produces a target sequence ρ. Obviously, the two events form a race iff $SeqCheck(\sigma, \rho, \mathcal{A})$ returns an event sequence.

Detect Deadlocks. Unlike data race, any deadlock occurs by following a sequence of events. Hence, we only need to check whether a feasible sequence (over lock acquisition events only) exists. For deadlocks of two threads, Algorithm 5 generates all two sequences where ρ_1 is shown in Figure 5(b). Obviously, the four events form a deadlock iff either of the call to $SeqCheck()$ returns an event sequence. For deadlocks of more threads, one can easily implement an algorithm to generate possible occurrence sequences and to check their feasibility accordingly.

Detect Atomicity Violations. Detection of Atomicity violation is even more straightforward. It is known that there are multiple patterns [34, 42, 56]. Algorithm 5 shows how to detect the pattern where a write event w_1 intrudes into a write-read pair (w_2, r). Given three events w_1, w_2, and r (suppose that they occur in this order in σ), the algorithm straightforward generates a sequence $\rho = \langle w_2, w_1, r\rangle$ and checks whether it is feasible, as illustrated in Figure 5(c). Note, Figure 5(c) shows two cases where the read event r can be from a third thread or from the thread $tid(w_2)$.

6 EVALUATION

6.1 Benchmarks and Traces

We collected a set of previously used Java benchmarks [3, 28, 37] where 31 programs were used for data race detection and atomicity violations detection. This set is almost the same set as evaluated before [29, 36, 37, 43] except 4 were not found. Only some of benchmarks in the AtomFuzzer paper are available, among which we included four, including the largest one. We used 20 benchmarks in Java from [26, 27] for deadlock detection. We run the tool Dirk [28] to generate traces.

The first six columns of Table 3 show the statistics of all traces including the numbers of threads, events, locations ("Mems"), locks, and branches. For well-readable purpose, we optimized the table, upper case "K" and "M" indicates thousand and million magnitude, respectively. We classify all traces into four categories according to the number of events ("n" in time complexity): **S-Bench** (<1M events), **M-Bench** (from 1M to 100M), **L-Bench** (from 100M to 1, 000M), and **XL-Bench** (>1, 000M events).

6.2 Experimental Setup

We implemented SeqCheck in Java and compared it with M2 [43], WCP [29], SHB [36], and SyncP [37] on data race detection. The four race detectors are published in recent years and SyncP is the-state-of-art. Section 7 has more discussion on them. They are available from the release package [37] . We compared SeqCheck with the sound deadlock detector Dirk [28] , which is also the state-of-the-art on sound deadlock detection except that Dirk is a constraint-solver-based one. For atomicity violation detection, we only found an available tool AtomFuzzer [41] for comparison. AtomFuzzer detects potential atomicity violations and then, for each of them, schedules a new execution with aim to trigger it.

All experiments were conducted on a Linux server with two Intel(R) Xeon(R) Gold-6148 CPUs and 256GiB RAM. Following the work [37], we setup a time limit of one hour, fairly limit the maximum usage of memory (80GiB) for each tools, run each tool one by one with guaranteeing no CPU/IO-dense processes running simultaneously. We conducted all experiment five times and took the average values. Different scheduling may produce different traces; we run the benchmarks once to collected the same set of traces.

Results on Data Race Detection. Table 3 shows the result on race detection, including the number of races, the time cost, and the max/mean distances of a race (i.e., the number of events between the two events). The symbols "−" and "*TO*" indicate the cases with no race detected and the cases where the time limit was reached, respectively. Note, for each approach, we collected and de-duplicated all reported races before it finished or run out of time. We use **SeqC** to denote SeqCheck in result tables from this subsection. For well-readable purpose, we use lower case "s", "m", "h" to indicate seconds, minutes, hours respectively.

From Table 3, we see that SeqCheck performed significantly better than others. On effectiveness, SeqCheck detected the largest set of races on each (group of) benchmark. We have manually confirmed that SeqCheck detected all races detected by others and no false positives were reported by SeqCheck on **S-Bench**. Overall, it detected 48 more races than the other four. Some of these races have a distance of more than 200M.

This shows the advantage of SeqCheck by analyzing branch events . Both SyncP and M2 detected a similar set of races (285 and 269). This is consistent with the previous result [37]. Both WCP and SHB detected a similar set of races (130 and 144).

On efficiency, SeqCheck spent 30 minutes on all benchmarks; while others spent from 6 hours to 11.5 hours. On each large benchmark (except **S-Bench**), SeqCheck is also the fastest one except on *montecarlo* and *series*. Overall, SeqCheck is nearly 12 times faster than all other approaches. Among the other four, M2 spent the most time (11.5 hours) and SyncP, WCP, and SHB spent from 6 hours to 7 hours. This result is also consistent with that of [37].

Another observation is that, except SeqCheck, all others reached 2 to 10 TO. And on the **XL-Bench** group, all run up to nearly or more than 1 hour except M2 on *h2*. But M2 has many more TO on all benchmarks. This result is consistent with the features of these four algorithms: the three (SyncP, WCP, SHB) are streaming algorithms and have almost a linear time complexity [37]; but M2 as well as SeqCheck is a full-trace algorithm where optimizations can be conducted for it.

Results On Deadlock Detection. Table 4 shows the results on deadlock detection by Dirk (with window size 10K) and SeqCheck. It also includes the number of threads and events.

On effectiveness, Dirk detected one more deadlock than SeqCheck. In detail, SeqCheck detected one additional deadlock (on *Vector*) missed by Dirk. All these are true positives based on

Table 3: Results on detection of data races.

	Benchmarks	Threads	Events	Mems	Locks	Branchs	#Races					Time					Distance	
							SyncP	M2	WCP	SHB	SeqC	SyncP	M2	WCP	SHB	SeqC	Max	Mean
S-Bench	array	3	47	4	1	9	-	-	-	-	-	0.03s	0.15s	0.03s	0.02s	0.10s	-	-
	critical	4	76	9	0	15	3	3	1	3	3	0.03s	0.09s	0.03s	0.02s	0.10s	29	17
	pingpong	8	189	16	0	57	1	1	1	1	1	0.03s	0.09s	0.03s	0.02s	0.07s	92	91
	airlinetickets	11	249	20	0	51	6	6	5	6	6	0.03s	0.14s	0.03s	0.02s	0.11s	161	127
	account	5	284	22	0	67	3	3	3	3	5	0.03s	0.10s	0.04s	0.02s	0.11s	152	109
	clean	12	505	41	2	130	1	3	1	1	3	0.08s	0.16s	0.05s	0.03s	0.11s	226	123
	bubblesort	12	2.3K	74	2	581	3	3	3	3	3	0.15s	0.71s	0.08s	0.05s	0.12s	1.2K	1.1K
	boundedbuffer	4	2.5K	39	4	871	8	8	6	8	10	0.07s	0.16s	0.08s	0.05s	0.08s	2.1K	1.1K
	mergesort	7	5.9K	594	6	2.2K	2	2	2	2	2	0.08s	0.17s	0.10s	0.07s	0.11s	2.4K	909
	raytracer	2	24.7K	3.9K	3	4.9K	4	4	4	4	5	0.23s	0.25s	0.29s	0.23s	0.09s	4.7K	2.1K
	bufwriter	5	27.9K	75	1	3.6K	4	4	4	4	4	0.63s	1.38s	0.46s	0.24s	0.18s	26.9K	6.8K
	ftpserver	5	99.8K	3.8K	71	56.9K	30	30	15	15	31	0.81s	5.17s	0.57s	0.41s	0.19s	36.2K	4.6K
	readerswriters	8	307.0K	27	1	121.4K	1	1	1	1	1	11.52s	1m33s	3.09s	1.07s	1.42s	1.2K	1.2K
	moldyn	5	555.0K	2.7K	1	148.0K	3	3	3	3	3	2.46s	20m54s	4.05s	2.07s	0.34s	57.0K	30.3K
M-Bench	jigsaw	13	2.8M	63.2K	104	930.0K	11	12	8	9	12	10.58s	17.92s	10.64s	10.09s	0.54s	44.3K	5.7K
	montecarlo	3	10.1M	850.1K	1	2.1M	2	2	1	2	2	38.38s	TO	47.15s	37.58s	40.97s	8.5M	4.2M
	sunflow	15	34.9M	2.0M	10	13.4M	7	7	5	6	7	1m13s	9m21s	2m18s	1m44s	5.33s	20.0M	3.0M
	crypt	17	79.0M	6.3M	1	6.8M	-	-	-	-	8	5m06s	TO	6m38s	5m07s	21.61s	56.8M	56.8M
L-Bench	eclipse	15	126.3M	10.4M	4.7K	62.1M	17	17	13	14	17	12m05s	18m32s	8m36s	6m24s	36.14s	18.3M	3.4M
	xalan	7	164.2M	2.8M	812	65.8M	128	135	9	9	138	TO	TO	9m51s	7m08s	1m56s	20.2M	13.1M
	lufact	5	179.6M	1.0M	1	48.7M	6	-	5	6	6	11m09s	TO	15m37s	10m37s	1m25s	5.1M	2.5M
	batik	7	279.9M	5.1M	1.9K	106.3M	-	-	-	-	-	11m41s	6m01s	15m23s	12m41s	22.59s	-	-
	lusearch	7	322.0M	4.7M	148	115.9M	16	-	16	16	19	14m21s	TO	17m51s	13m46s	31.16s	200.6M	12.8M
	pmd	9	382.6M	12.1M	221	168.9M	23	23	13	16	24	16m07s	TO	20m28s	16m20s	35.91s	224.4M	33.6M
	tsp	5	487.1M	180.9K	2	167.3M	4	-	4	4	4	18m36s	TO	22m25s	17m15s	2m20s	86.7M	53.0M
	series	18	574.2M	286.4K	1	573.1M	-	-	-	-	-	10.12s	14.65s	11.00s	10.19s	37.17s	-	-
	luindex	3	775.0M	2.5M	65	304.6M	1	1	1	1	1	27m48s	14m35s	35m56s	27m36s	1m02s	1.9K	1.9K
XL-Bench	sparsematmult	6	1,286.9M	16.0M	1	150.3M	-	-	-	-	-	TO	TO	TO	TO	5m07s	-	-
	sor	5	1,357.3M	4.0M	1	187.3M	-	-	-	-	10	TO	TO	TO	TO	4m08s	40.1M	17.6M
	avrora	7	1,636.3M	864.5K	6	638.4M	-	-	6	6	7	TO	TO	TO	58m03s	4m52s	896.8K	364.4K
	h2	2	2,088.7M	27.1M	15	1,126.0M	1	1	-	1	1	TO	25m19s	TO	59m31s	5m16s	2	2
	Total	-	**9,787.9M**	**96.4M**	**8.1K**	**3,738.2M**	**285**	**269**	**130**	**144**	**333**	**>6h59m**	**>11h36m**	**>6h36m**	**>5h57m**	**30m03s**	-	-

our code inspection. On *Vector*, the distance of the deadlock (i.e., the two acquire events) is 2.7M which is very large window and constraint-solver-based approaches [24] are probably unable to detect. On all other benchmarks, the distance is at most 1.6K. Hence, Dirk was able to detect two deadlocks on *Deadlock* and *Transfer* missed by SeqCheck. On these two deadlocks, there are data flows that can be handled by Dirk but not SeqCheck. It is challenging to extend SeqCheck to handle these cases; we leave it as a further work. Note, some benchmarks have deadlocks but their traces do not have one [28]; and both tools did not detect deadlocks on them.

On efficiency, SeqCheck is significantly better than Dirk. SeqCheck spent less than 6 seconds in total and less than 2.5 seconds on each benchmark. However, Dirk reached one TO and took much more time, especially on benchmarks with >500K events.

Results On Atomicity Violations Detection. We configured SeqCheck to detect the atomicity violation pattern $\langle w_1, w_2, r\rangle$ as that shown in Algorithm 5 and we set the distance between w_2 and r to be at most 100. The result is shown in Table 5, including the number of atomicity violations, the time cost, and the max/mean distance. We use "⊗" to indicate a crash in testing.

The result shows that SeqCheck finished in less than 20 minutes and 30 unique ones were detected. Some of these atomicity violations have a distance up to 18.7M and SeqCheck finished in about 20 seconds. This shows that SeqCheck is efficient on detecting atomicity violations. The results on **S-Bench** are also manually confirmed and they match the pattern $\langle w_1, w_2, r\rangle$.

AtomFuzzer (i.e., "**AtomF**") detected none on all benchmarks and it frequently crashed on many benchmarks. We have already tried our best to avoid crashes as much as possible. But, it seems no avail. The main reason of crashes is that AtomFuzzer does not support Java reflection.

7 RELATED WORKS

Traditional Unsound Approaches. Two earliest works on data race detection are based on the happens-before relation [31] and the lockset discipline [51]. The former define a partial order over synchronizations and is widely adopted in many data race detectors [1, 2, 4, 10, 12, 13, 15, 20, 45, 52–54, 57]. The latter defines a data race if two accesses are not protected by a common lock [9, 39, 47, 51, 60]. Hybrid analyses combine the two approaches to improve accuracy

Table 4: Results on detection of deadlocks.

Benchmarks	Threads	Events	#Deadlocks		Time		Distance	
			Dirk	SeqC	Dirk	SeqC	Max	Mean
Deadlock	3	50	1	-	0.02s	0.07s	-	-
NotADlk	3	62	-	-	0.02s	0.10s	-	-
Picklock	3	68	1	1	0.02s	0.07s	25	25
HashTable	3	70	2	2	0.03s	0.10s	41	36
Bensalem	4	80	1	1	0.06s	0.07s	29	29
Transfer	3	81	1	-	0.02s	0.10s	-	-
Test	3	83	2	2	0.06s	0.07s	34	27
StringBuffer	3	116	2	2	0.03s	0.07s	34	19
DiningPhi	3	166	1	1	0.05s	0.10s	105	105
DirkAccount	6	867	-	-	0.15s	0.10s	-	-
Log4j2	4	2.5K	1	1	0.80s	0.10s	295	295
Dbcp1	3	2.9K	2	2	0.23s	0.10s	210	191
Derby2	3	3.0K	1	1	0.31s	0.12s	61	61
Dbcp2	3	4.1K	-	-	0.55s	0.10s	-	-
JDBC 1	3	674.5K	2	2	44.92s	0.19s	424	240
JDBC 2	3	674.6K	1	1	4.33s	0.19s	168	168
JDBC 3	3	675.8K	1	1	43.48s	0.16s	148	148
JDBC 4	3	675.9K	2	2	10.44s	0.20s	1.6K	1.6K
Vector	3	5.4M	-	1	TO	1.34s	2.7M	2.7M
TestPerf	10	11.9M	-	-	48.63s	2.47s	-	-
Total	**-**	**20.0M**	**21**	**20**	**>1h02m**	**5.95s**	-	-

[11, 40]. Others include scheduling [5] and sampling approaches [4, 7, 22]. Unfortunately, the above can report false positives.

Sound Offline Analysis. Sound dynamic ones, as discussed in Section 1, include three types. M2 is a graph based one that has been extensively discussed. Dirk [28] and RVPredict [24] are representatives of constraint solver based ones. They can infer alternative executions by considering branches and concrete read/write values; hence, they have the potential to cover many races, as well as deadlocks [28] and other concurrency bugs (like concurrency Use-after-free and concurrency Null-pointer-dereference [16, 23]). However, they rely on (e.g., SMT) solvers. This prevents them from analyzing a full execution trace. They can miss a race with much a larger distance.

Sound Online Analysis. In recent years, more and more sound dynamic online approaches have been proposed like CP [55], WCP [29], SHB [36],DC [50], SDP/WDP [21], and SyncP [29]. These approaches track dependency among memory events and guarantee that the predicted (partial) trace is feasible via vector clocks [31]. HB and SHB both miss simple races because they cannot swap the critical sections. The other approaches based on HB construct weaker partial orders in order to reduce the degree of incompleteness. CP and WCP are sound but incomplete even for two threads. DC and SDP are unsound weakenings of WCP; and WDP is an unsound weakening of DC. The DC/WDP-races filtered by a vindication algorithm become sound but incomplete [21, 50]. The recently introduced SyncP is the state of the art in detecting races using online techniques. All of these online approaches are computable in linear time and have been compared in our experiments.

Other approaches such as static race analysis [14, 38, 46, 48, 58, 61] are unsound, reporting false races. Techniques such as [17, 49, 59] implement efficient dynamic race detectors. Tools such as RoadRunner[19] and Rapid [35] provide dynamic analysis frameworks to facilitate experimentation for concurrent programs.

Table 5: Full Results on Detection of Atomicity Violation.

	Benchmarks	#Atom		Time		Distance	
		AtomF	SeqC	AtomF	SeqC	Max	Mean
S-Bench	array	-	-	0.17s	0.07s	-	-
	critical	-	1	0.07s	0.10s	20	13
	pingpong	-	-	0.76s	0.07s	-	-
	airlinetickets	-	1	0.07s	0.10s	159	159
	account	-	1	0.56s	0.10s	118	118
	clean	-	-	0.07s	0.07s	-	-
	bubblesort	-	-	0.08s	0.11s	-	-
	boundedbuffer	-	2	0.17s	0.08s	812	761
	mergesort	-	-	0.07s	0.11s	-	-
	raytracer	-	1	0.12s	0.08s	1.1K	1.1K
	bufwriter	-	-	0.08s	0.15s	-	-
	ftpserver	⊗	1	⊗	0.17s	1.8K	1.8K
	readerswriters	-	-	0.40s	0.73s	-	-
	moldyn	-	-	0.19s	0.30s	-	-
M-Bench	jigsaw	⊗	1	⊗	0.37s	111	111
	montecarlo	⊗	-	⊗	48.72s	-	-
	sunflow	⊗	-	⊗	4.22s	-	-
	crypt	-	-	0.59s	20.32s	-	-
L-Bench	eclipse	⊗	-	⊗	13.62s	-	-
	xalan	⊗	20	⊗	20.86s	18.7M	12.8M
	lufact	-	-	5.72s	48.31s	-	-
	batik	⊗	-	⊗	25.10s	-	-
	lusearch	⊗	-	⊗	28.09s	-	-
	pmd	⊗	-	⊗	35.94s	-	-
	tsp	⊗	-	⊗	46.30s	-	-
	series	-	-	26m27s	38.07s	-	-
	luindex	⊗	-	⊗	54.61s	-	-
XL-Bench	sparsematmult	-	-	23.27s	3m16s	-	-
	sor	-	-	9.01s	3m21s	-	-
	avrora	⊗	2	⊗	2m54s	141.0K	130.4K
	h2	⊗	-	⊗	3m07s	-	-
	Total	**-**	**30**	**27m08s**	**19m07s**	**18.9M**	**12.9M**

8 CONCLUSION

We have presented an efficient, sound dynamic approach SeqCheck for detection of general concurrency bugs. It advanced M2 by modeling branch events and supporting decisions on whether an event sequence is feasible. With SeqCheck, one can easily encode a potential concurrency bug into one or more sequences of events. SeqCheck has built in the sequence generation for data races, deadlocks, and atomicity violations. The experimental results show that SeqCheck achieved significantly better results than recent sound data race and deadlock detectors in both effectiveness and efficiency.

ACKNOWLEDGEMENTS

We sincerely thank the anonymous reviewers for helpful suggestions and insights for improving this paper. This work is supported in part by National Natural Science Foundation of China (NSFC) (Grant No. 61932012), the Key Research Program of Frontier Sciences, CAS (Grant No. ZDBS-LY-7006), the Youth Innovation Promotion Association of the Chinese Academy of Sciences (YICAS) (Grant No. 2017151), and the National Key Research and Development Program of China (No. 2018YFB1403400). And National Science Foundation award 1815496 supported Jens Palsberg.

REFERENCES

[1] Swarnendu Biswas, Man Cao, Minjia Zhang, Michael D. Bond, and Benjamin P. Wood. 2017. Lightweight data race detection for production runs. In *Proceedings of the 26th International Conference on Compiler Construction* (Austin, TX, USA) *(CC'17)*. Association for Computing Machinery, ustin, TX, USA, 11–21. https://doi.org/10.1145/3033019.3033020

[2] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. 2015. Valor: efficient, software-only region conflict exceptions. *ACM SIGPLAN Notices* 50, 10 (Oct. 2015), 241–259. https://doi.org/10.1145/2858965.2814292

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[4] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. 2010. PACER: Proportional Detection of Data Races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 255–268. https://doi.org/10.1145/1806596.1806626

[5] Yan Cai and Lingwei Cao. 2015. Effective and Precise Dynamic Detection of Hidden Races for Java Programs *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 450–461. https://doi.org/10.1145/2786805.2786839

[6] Yan Cai and W. K. Chan. 2012. MagicFuzzer: Scalable Deadlock Detection for Large-scale Applications. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 606–616. http://dl.acm.org/citation.cfm?id=2337223.2337294

[7] Yan Cai, Jian Zhang, Lingwei Cao, and Jian Liu. 2016. A Deployable Sampling Strategy for Data Race Detection. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 810–821. https://doi.org/10.1145/2950290.2950310

[8] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. 2019. Detecting Concurrency Memory Corruption Vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 706–717. https://doi.org/10.1145/3338906.3338927

[9] Jong-Deok Choi, Keunwoo Lee, and Alexey Loginov. 2002. Efficient and precise datarace detection for multithreaded object-oriented programs. *ACM Sigplan Notices* 37, 5 (June 2002), 258–269. https://doi.org/10.1145/543552.512560

[10] Intel Corporation. 2016. Intel Inspector. https://software.intel.com/en-us/intel-inspector-xe

[11] Anne Dinning and Edith Schonberg. 1991. Detecting access anomalies in programs with critical sections. *ACM SIGPLAN Notices* 26, 12 (Dec. 1991), 85–96. https://doi.org/10.1145/127695.122767

[12] Laura Effinger-Dean, Brandon lucia, luis Ceze, Dan Grossman, and Hans-J Boehm. 2012. IFRit: interference-free regions for dynamic data-race detection. In *Acm International Conference on Object Oriented Programming Systems Languages & Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. https://doi.org/10.1145/2398857.2384650

[13] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. 2007. Goldilocks: a race and transaction-aware java runtime. *Acm Sigplan Notices* 42, 6 (June 2007), 245–255. https://doi.org/10.1145/1273442.1250762

[14] Dawson Engler and Ken Ashcraft. 2003. RacerX : Effective, static detection of race conditions and deadlocks. *ACM SIGOPS Operating Systems Review* 37, 5 (Oct. 2003), 237–252. https://doi.org/10.1145/945445.945468

[15] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI '10)*. 151–162. https://doi.org/10.5555/1924943.1924954

[16] Azadeh Farzan, P. Madhusudan, Niloofar Razavi, and Francesco Sorrentino. 2012. Predicting Null-Pointer Dereferences in Concurrent Programs. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 47, 11 pages. https://doi.org/10.1145/2393596.2393651

[17] Mingdong Feng and Charles E. Leiserson. 1997. Efficient detection of determinacy races in Cilk programs. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures (SPAA '97)*. 1–11.

[18] Peter M. Fenwick. 1994. A New Data Structure for Cumulative Frequency Tables. *Softw. Pract. Exper.* 24, 3 (March 1994), 327–336. https://doi.org/10.1002/spe.4380240306

[19] Cormac Flanagan and Stephen Freund. 2010. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '10)*. 1–8.

[20] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. ACM, New York, NY, USA, 121–133. https://doi.org/10.1145/1542476.1542490

[21] Kaan Genç, Jake Roemer, Yufan Xu, and Michael D. Bond. 2019. Dependence-Aware, Unbounded Sound Predictive Race Detection. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 179 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360605

[22] Yu Guo, Yan Cai, and Zijiang Yang. 2017. AtexRace: Across Thread and Execution Sampling for in-House Race Detection. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 315–325. https://doi.org/10.1145/3106237.3106242

[23] Jeff Huang. 2018. UFO: Predictive Concurrency Use-after-Free Detection. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 609–619. https://doi.org/10.1145/3180155.3180225

[24] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

[25] Joab Jackson. 2012. Nasdaq's Facebook glitch came from 'race conditions'. http://www.computerworld.com/s/article/9227350.

[26] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 2009. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 110–120. https://doi.org/10.1145/1542476.1542489

[27] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. 2008. Deadlock Immunity: Enabling Systems to Defend against Deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 295–308.

[28] Christian Gram Kalhauge and Jens Palsberg. 2018. Sound Deadlock Prediction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 146 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276516

[29] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic Race Prediction in Linear Time *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

[30] Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* C-28, 9 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

[31] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[32] N. G. Leveson and C. S. Turner. 1993. An investigation of the Therac-25 accidents. *Computer* 26, 7 (1993), 18–41. https://doi.org/10.1109/MC.1993.274940

[33] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. Association for Computing Machinery, New York, NY, USA, 329–339. https://doi.org/10.1145/1346281.1346323

[34] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. 2008. Atom-Aid: Detecting and Surviving Atomicity Violations. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, USA, 277–288. https://doi.org/10.1109/ISCA.2008.4

[35] Umang Mathur. 2020. RAPID : Dynamic Analysis for Concurrent Programs. https://github.com/umangm/rapid

[36] Umang Mathur, Dileep Kini, and Mahesh Viswanathan. 2018. What Happens-after the First Race? Enhancing the Predictive Power of Happens-before Based Dynamic Race Detection. 2, OOPSLA, Article 145 (Oct. 2018), 29 pages. https://doi.org/10.1145/3276515

[37] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2021. Optimal Prediction of Synchronization-Preserving Races. *Proc. ACM Program. Lang.* 5, POPL, Article 36 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434317

[38] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. *ACM SIGPLAN Notices* 41, 6 (June 2006), 308–319. https://doi.org/10.1145/1133255.1134018

[39] Hiroyasu Nishiyama. 2004. Detecting Data Races Using Dynamic Escape Analysis Based on Read Barrier. In *Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium (VM '04)*. New York, NY, USA, 127–138. https://doi.org/10.5555/1267242.1267252

[40] Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming* (San Diego, California, USA) *(PPoPP'03)*. New York, NY, USA, 167–178. https://doi.org/10.1145/966049.781528

[41] Chang-Seo Park and Koushik Sen. 2008. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. Association for Computing Machinery, New York, NY, USA, 135–145. https://doi.org/10.1145/1453101.1453121

[42] Chang-Seo Park and Koushik Sen. 2008. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia) *(SIGSOFT '08/FSE-16)*. Association for Computing Machinery, New York, NY, USA, 135–145. https://doi.org/10.1145/1453101.1453121

[43] Andreas Pavlogiannis. 2020. Fast, sound, and effectively complete dynamic race prediction. *Proc. ACM Program. Lang.* 4, POPL (2020), 17:1–17:29. https://doi.org/10.1145/3371085

[44] Kevin Poulsen. 2012. Software bug contributed to blackout. Security Focus. http://www.securityfocus.com/news/8016.

[45] Eli Pozniansky and Assaf Schuster. 2007. Multirace: Efficient On-the-fly Data Race Detection In Multithreaded C++ Programs. *ACM Trans. Comput. Syst.* 19, 3 (Nov. 2007), 327–340. https://doi.org/10.1002/cpe.1064

[46] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. 2011. LOCKSMITH: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems* 33, 1 (Jan. 2011). https://doi.org/10.1145/1889997.1890000

[47] Christoph von Praun and Thomas R. Gross. 2001. Object Race Detection. *ACM Sigplan Notices* 36, 11 (Nov. 2001), 70–82. https://doi.org/10.1145/504311.504288

[48] Cosmin Radoi and Danny Dig. 2013. Practical static race detection for Java parallel loops. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*. 178–190.

[49] Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and precise dynamic datarace detection for structured parallelism. *ACM SIGPLAN Notices* 47, 6 (June 2012), 531–542. https://doi.org/10.1145/2345156.2254127

[50] Jake Roemer, Kaan Genc, and Michael D. Bond. 2018. High-Coverage, Unbounded Sound Predictive Race Detection *(PLDI 2018)*. 374–389. https://doi.org/10.1145/3192366.3192385

[51] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

[52] Koushik Sen. 2008. Race directed random testing of concurrent programs. In *ACM SIGPLAN Notices* (Tucson, Arizona, USA) *(PLDI '08)*. 11–21. https://doi.org/10.1145/1379022.1375584

[53] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, NY, USA) *(WBIA '09)*. 62–71. https://doi.org/10.1145/1791194.1791203

[54] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. 2012. Dynamic Race Detection with LLVM Compiler. In *Runtime Verification (RV 2011)*. 110–114. https://doi.org/10.1007/978-3-642-29860-8_9

[55] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

[56] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. 2010. PENELOPE: Weaving Threads to Expose Atomicity Violations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. Association for Computing Machinery, New York, NY, USA, 37–46. https://doi.org/10.1145/1882291.1882300

[57] Kaushik Veeraraghavan, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. 2011. Detecting and surviving data races using complementary schedules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*. 369–384. https://doi.org/10.1145/2043556.2043590

[58] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE '07)*. 205–214.

[59] Adarsh Yoga, Santosh Nagarakatte, and Aarti Gupta. 2016. Parallel data race detection for task parallel programs with locks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. 833–845.

[60] Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom) *(SOSP '05)*. Association for Computing Machinery, New York, NY, USA, 221–234. https://doi.org/10.1145/1095810.1095832

[61] Sheng Zhan and Jeff Huang. 2016. ECHO: instantaneous in situ race detection in the IDE. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. 775–786.